LECTURE 3
PERFORMANCE ANALYSIS

SEHH2239 Data Structures

LEARNING OBJECTIVES:

- To understand the use of algorithm analysis
- To assess the efficiency of a given algorithm
- To compare the expected execution time of different algorithms

ALGORITHMS AND PSEUDO CODES

WHAT IS AN ALGORITHM?

- An algorithm is a step by step method of solving a problem.
  - E.g. Find the path from home to the campus. Cook a streamed fish, etc.
- It is commonly used for data processing, calculation and other related computer and mathematical operations.*

*Quote from: https://www.techopedia.com/definition/3739/algorithm

PSEUDOCODE

- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

ALGORITHMS AND PSEUDOCODES

- Pseudocode is a *step-by-step written outline* of your code that you can gradually transcribe into the programming language.
- *Describing how an algorithm should work.*
- Pseudocode can illustrate where a particular construct, mechanism, or technique could or must appear in a program.

https://www.wikihow.com/Write-Pseudocode

PSEUDOCODE AND ACTUAL CODE

Pseudocode

If age is 65 or above
        Group is senior
Else if age is 18 or above
        Group is adult
Else
        Group is children
Display Group

In JAVA

```
if (age >= 65)
        group = "senior";
else if (age >= 18)
        group = "adult";
else
        group = "children";
System.out.println(group);
```

Actual Code

In PYTHON

```
if age >= 65:
  group = "senior"
else:
  if age >= 18:
      group = "adult"
  else:
    group = "children"
print(group)
```

PSEUDOCODE DETAILS

- Control flow
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …
  - Indentation replaces braces
- Method declaration

  Algorithm *method* (*arg* [, *arg*…])

  Input …

  Output …

- Method call

  var.method (*arg* [, *arg*…])

- Return value

  return *expression*

- Expressions

  $\leftarrow$ Assignment (like = in Pyhton)

  $=$ Equality testing (like == in Pyhton)

  n^2 Superscripts and other mathematical formatting allowed

EFFICIENCY OF ALGORITHMS

ALGORITHM EFFICIENCY

Problem:

Total = 1+2+3…+N.

where N is any +ve integer

Find Total.

Algorithm A

```
total = 0
for i : 1 to N
    total = total + i
```

Algorithm B

```
total = 0
for i : 1 to N
    for m : 1 to i
     total = total + 1
```

Algorithm C

```
total = N *( N + 1 ) / 2
```

- Which one runs fastest? Which slowest?

MEASURING AN ALGORITHM EFFICIENCY

- How to measure efficiency to compare different algorithms to solve a problem?
- The process of measuring the *complexity* of algorithms is called ***analysis of algorithms***.

Complexity

- Time Complexity – The time it takes to execute
- Space Complexity – The memory it needs to execute

- Each of them can be analyzed separately.
  - Focus on the time complexity of algorithms.
    - As more important, and memory size grows exponentially.
  - Inverse relation between time and space required.

PROBLEM SIZE

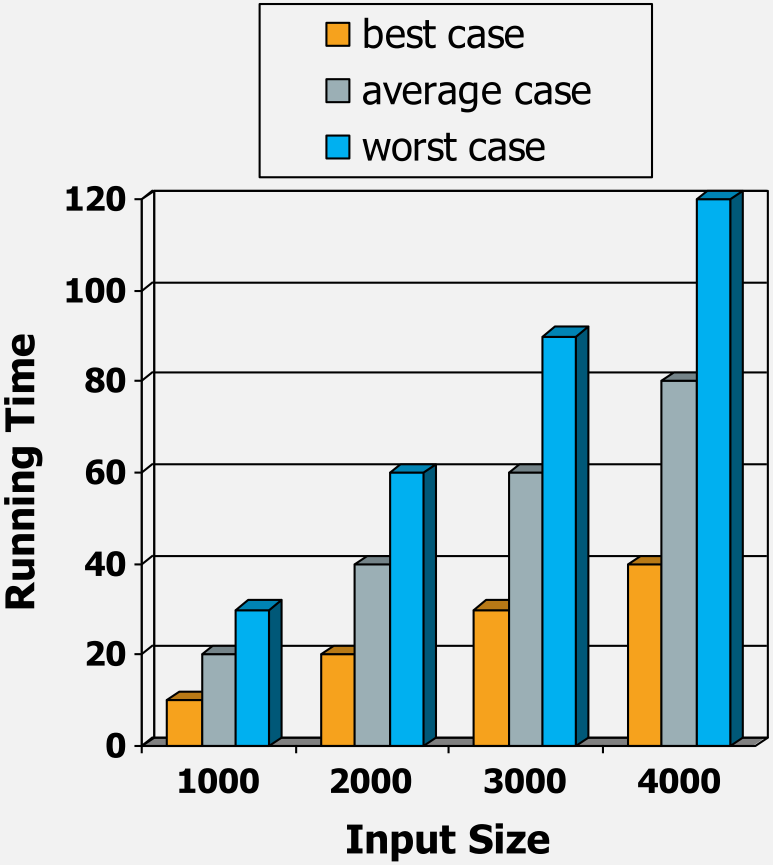
- Problem size is the *number of items* that an algorithm processes.
  - E.g. Number of elements in a list.
- The *running time* of an algorithm typically grows with the input size.
  - E.g. Time needed for removing a elements grows with the number of elements in a list.

BEST, WORST AND AVERAGE CASES OF RUNNING TIME

- Best-case
  - The algorithm takes the least time and it can do no better than that.
- Worst-case
  - The algorithm takes the most time and it can do no worse than that.
- Average-case
  - The average case on take typical data.

RUNNING TIME

- Average case time is often difficult to determine.
- We focus on the **worst case** running time.
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics
  - Worst-case count = maximum count



Big-Oh Notation

BIG-OH NOTATION

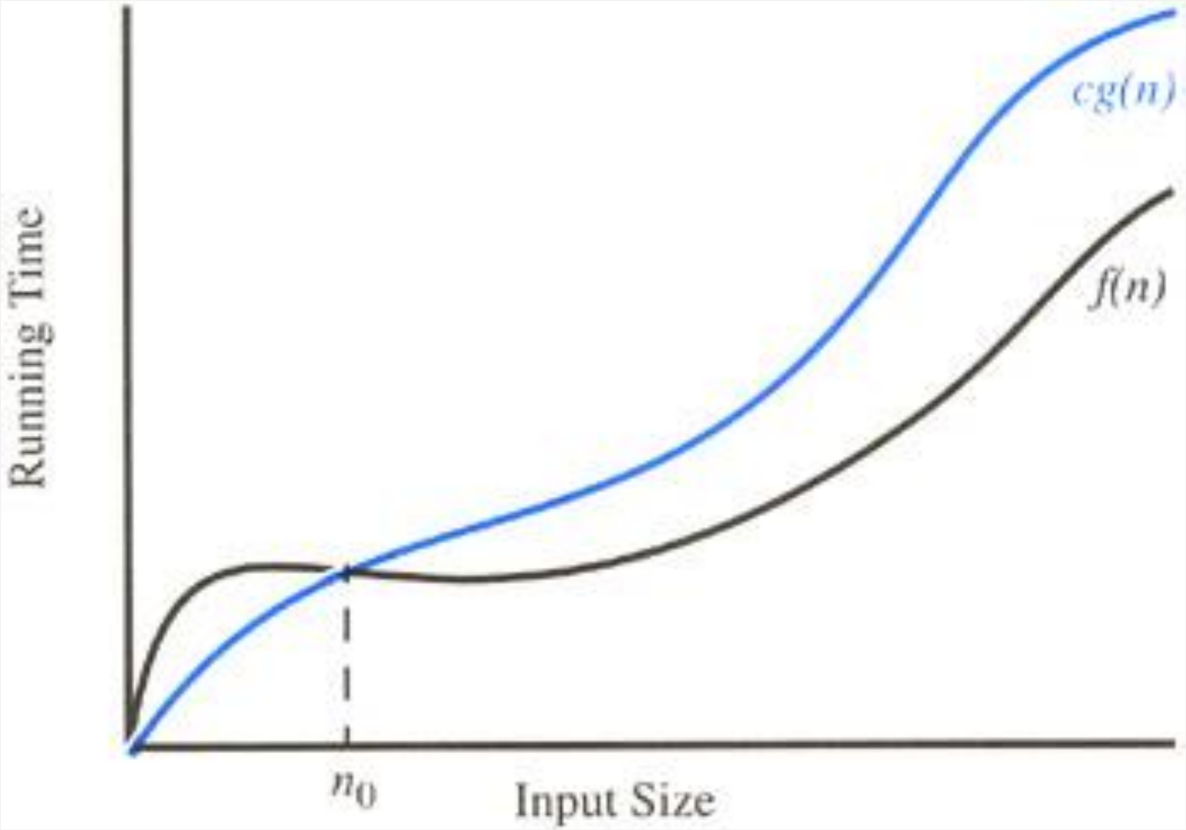
- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that

$$f(n) \leq cg(n) \quad \text{for } n \geq n_0$$

$c \times g(n)$ gives the **upper-bound** on f(n).

$f(n) \leq O(g(n))$

BIG-OH NOTATION



Illustrating the "big-Oh" notation. The function f(n) is O(g(n)), for f(n) ≤ c·g(n) when n ≥ n_0

BIG-OH NOTATION

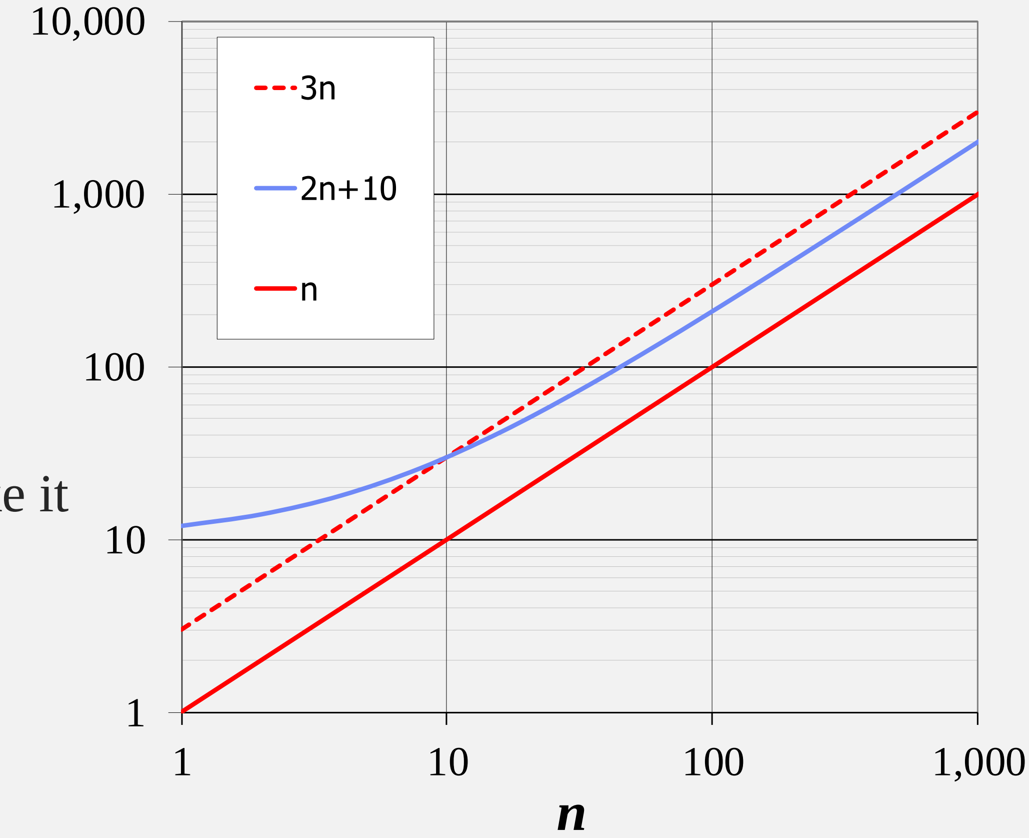
Example:

- $2n + 10$ is $O(n)$

  Because $2n + 10 \leq cn$

  for $c = 3$ and $n_0 = 10$

  (That is, there are positive constants c and n_0 to make it true.)



BIG-OH NOTATION

$O(n)$

- We read O(*n*) as either "Big Oh of *n*" or "order of at most *n*".

$f(n) = 6n+3 \quad | \quad f(n) \leq 7n$ for $n \geq n_0 = 2$

$\underbrace{g(n)} = 7n$ $\quad c = 7$

Example

- If an algorithm uses 6n+3 operations, it requires time proportional to *n*. We say it is O(n).
- If an algorithm has a time requirement of proportional to n^2, we say that it is $O(n^2)$.

MORE BIG-OH EXAMPLES

- $7n - 2 \le 7n \quad n \ge n_0 = 1$

  $c = 7$ $\quad O(n)$

  7n-2 is O(n)

  need $c > 0$ and $n_0 \ge 1$ such that $7n - 2 \le cn$ for $n \ge n_0$

  this is true for $c = 7$ and $n_0 = 1$

- $3n^3 + 20n^2 + 5n^0 \le 3n^3 + 20n^3 + 5n^3$

  $= 28(n^3)$ for $n_0 \ge 1$

  $3n^3 + 20n^2 + 5$ is $O(n^3)$

  need $c > 0$ and $n_0 \ge 1$ such that $3n^3 + 20n^2 + 5 \le cn^3$ for $n \ge n_0$

  this is true for $c = 4$ and $n_0 = 21$

  $O(n^3)$

- $3\log_2 n + 5 \le 3\log n + 5\lg n \quad n \ge n_0 = 2$

  $c = 8$ $\quad O(\lg n)$

  3 log n + 5 is O(log n)

  need $c > 0$ and $n_0 \ge 1$ such that $3\log n + 5 \le c\log n$ for $n \ge n_0$

  this is true for $c = 8$ and $n_0 = 2$

BIG-OH RULES

- If is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,

> 1. **Drop lower-order terms**
> 2. **Drop constant factors**

$$ax^3 + bx^2 + cx + dx^0$$

$$\cancel{a}x^3$$

- Use the smallest possible class of functions
    - "$2n$ is $O(n)$" $O(n^2)$

$$O(x^3)$$

- Use the simplest expression of the class
    - "$3n + 5$ is $O(n)$" $O(3n)$

BIG-OH EXAMPLES

$20n^3 + 10n\log n + 5$ *is* $O(n^3)$.

Justification: $20n^3 + 10n\log n + 5 \leqslant 35n^3$, *for* $n \geqslant 1$.

In fact, any polynomial $a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0$ will always be $O(n^k)$.

$3\log n + \log\log n$ *is* $O(\log n)$.

Justification: $3\log n + \log\log n \leqslant 4\log n$, *for* $n \geqslant 2$. *Note that* $\log\log n$ *is not even defined for* $n = 1$. *That is why we use* $n \geqslant 2$.

2^{100} *is* $O(1)$.

Justification: $2^{100} \leqslant 2^{100} \cdot 1$, *for* $n \geqslant 1$. *Note that variable n does not appear in the inequality, since we are dealing with constant-valued functions.*

$5/n$ *is* $O(1/n)$.

Justification: $5/n \leqslant 5(1/n)$, *for* $n \geqslant 1$ *(even though this is actually a* **decreasing** *function).*

FIND THE BIG-OHS

$$10n + 7$$

$$100n^3 - 3$$

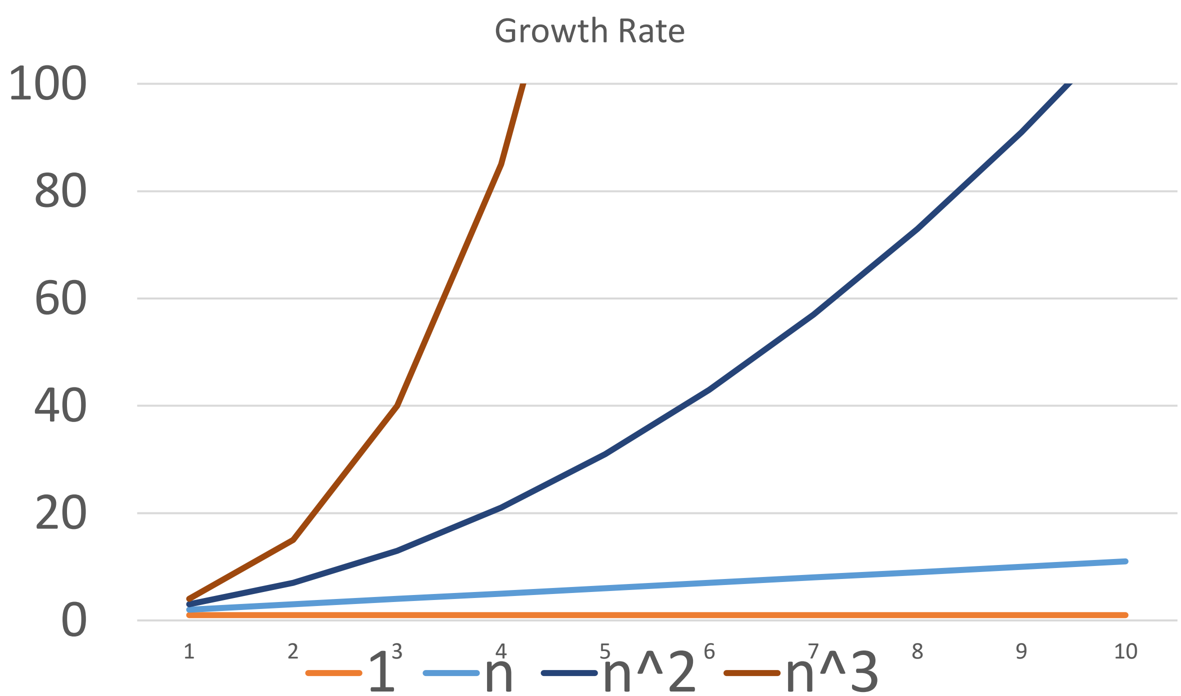
$$3n^2 + 2n + 6$$

$$8n^4 + 9n^2$$

GROWTH RATE FUNCTIONS

GROWTH RATE

- Relation to the problem size, n
- We care about *n* to be very large.
- Growth rate :



GROWTH RATE

- Functions

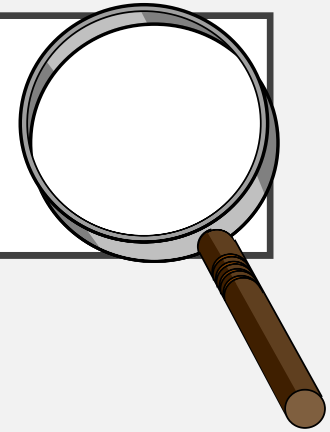
$$1 < \log(\log n) < \log(n) < n$$

$$n < n \log n < n^2 < n^3 < 2^n < n!$$

- Note that log here are base 2

ANALYSIS OF ALGORITHM EFFICIENCY IN BIG-OH NOTATION

THEORETICAL ANALYSIS



- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, *n*
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

ESTIMATING RUNNING TIME

- The meaning of Big-Oh functions in running time.
  - O(1) is constant time.
  - O(n) is linear time.
  - O(n^2) is quadratic time.
  - O(log n) is logarithmic time.
  - ….
- Useful tricks:
  - Investigate nesting of loops due to inputs.
  - Look into the steps in a loop due to inputs.
- Focus in the worst case.

ALGORITHM EFFICIENCY

Problem:

Total = 1+2+3…+N.

Find total.

Algorithm A

```
total = 0
for i : 1 to N
    total = total + i
```

O(n)

Algorithm B

```
total = 0
for i : 1 to N
   for m : 1 to i
    total = total + 1
```

O(n^2)

Algorithm C

```
total = N *( N + 1 ) / 2
```

O(1)

EXAMPLE

```
for i in range(0, N):
    for j in range(N, i, -1):
        a = a + i + j
```

https://realpython.com/lessons/time-complexity-overview/

https://www.youtube.com/watch?v=5yJ_QLec0Lc

COMPLEXITY OF PYTHON LIST

- The complexity of Python List operations are listed in Table below.

Operation	Example	Complexity Class	Notes
Index	`l[i]`	O(1)	
Store	`l[i] = 0`	O(1)	
Length	`len(l)`	O(1)	
Append	`l.append(5)`	O(1)	
Pop	`l.pop()`	O(1)	same as l.pop(-1), popping at end
Clear	`l.clear()`	O(1)	similar to l = []
Slice	`l[a:b]`	O(b-a)	l[1:5]:O(l)/l[:]:O(len(l)-0)=O(N)
Extend	`l.extend(...)`	O(len(...))	depends only on len of extension
Construction	`list(...)`	O(len(...))	depends on length of ... iterable
check ==, !=	`l1 == l2`	O(N)	
Insert	`l[a:b] = ...`	O(N)	
Delete	`del l[i]`	O(N)	
Containment	`x in/not in l`	O(N)	searches list
Copy	`l.copy()`	O(N)	Same as l[:] which is O(N)
Remove	`l.remove(...)`	O(N)	
Pop	`l.pop(i)`	O(N)	O(N-i): l.pop(0):O(N) (see above)
Extreme value	`min(l)/max(l)`	O(N)	searches list
Reverse	`l.reverse()`	O(N)	
Iteration	`for v in l:`	O(N)	

http://tp60011275.blogspot.com/2018/12/performance-of-python-data-structures.html

SUMMARY

- An algorithm's complexity is described in terms of the time and space required to execute it.
- Compare efficiency of algorithms
- Big-Oh Notation
- Growth-rate function